Programmieren in C++

Wintersemester 2024/25

Prof. Armin Scrinzi

Zweck der Vorlesung

Ausgehend von bestehenden Programmierkenntnissen

die nötigen Skills

- für komplexere eigenständige Programmieraufgaben
- zur selbständigen Erweiterung der Skills
- um Beiträge zu bestehenden Projekten zu leisten

Organisation und Ablauf

Characteristics of the course

Write a (moderately) complex code for solving the time-dependent Schrödinger equation

Discuss all aspects of programming and computing as that example advances

The course is not a comprehensive formal course on C++

The actual practice of programming:

Good programming style

Maintainability

Portability

Collaboration

Debugging

Ensuring correctness

Using available software

Contents

- C++ Basics (quick guided tour through W3schools tutorial on C++)
- Example application: solution of the time-dependent Schrödinger equation
- Inheritance: base, derived, and abstract classes
- Using private/public/protected
- gdb - debugging
- Clean code (e.g. Google style guide)
- Cmake – building complex projects
- git – version control
- Templates
- Code efficiency
- Pointers and memory management
- Parallelizing with OpenMP and MPI
- Recursive and tree-structures

...

Klausur

Form schriftlich auf Papier wie “Einführung ins Programmieren”

Aufgaben werden dem Niveau der VO schwieriger

Fragen

???

Introduction

Programming languages

Common for numerical applications in physics

C++

Many large code projects,
default choice for new projects

Fortran

Many large code projects,
THE physics language of the 20th century

Python

Wide use for scripting, plotting, AI

Julia (similar to python)

Matlab

Frequent use for quick calculations
and prototyping of numerical methods

C++ vs. Python

	C++	Python
Compilation	Compiled	Interpreted
Usage	Not easy	Really easy
Types	Static	Dynamic
Prototyping	Hard (-ly possible)	easy
Performance	maximal	slow

Guiding project for this lecture

The TDSE - time-dependent Schrödinger equation

TDSE – the time-dependent Schrödinger equation

Hydrogen atom in a laser field (in “atomic units”)

$$i\frac{d}{dt}\Psi(\vec{r},t) = \left[\left(-i\vec{\nabla} + \vec{A}(t)\right)^2 - \frac{1}{r}\right]\Psi(\vec{r},t)$$

...don’t worry, things will be explained in due time…
Physics in T2 (some time around Christmas)

i	imaginary unit
t	time
$\vec{r} = (x, y, z)$	Cartesian coordinates
$\Psi(\vec{r},t)$	“wave function”, complex valued, all info about atom
$\vec{A}(t) = (A_x(t), A_y(t), A_z(t))$	“vector potential”, describes Laser field
$\vec{\nabla} = \left(\frac{d}{dx}, \frac{d}{dy}, \frac{d}{dz}\right)$	“Nabla operator”
$-\frac{1}{r} := -\frac{1}{\lvert\vec{r}\rvert}$	“atomic potential” binds electron to proton
$-\left(\vec{\nabla} - \vec{A}(t)\right)^2 - \frac{1}{r} := \hat{H}(t)$	“Hamilton operator” – all about the dynamics

The TDSE in one dimension

Hydrogen atom in a laser field in 1d

$$i\frac{d}{dt}\Psi(x,t) = \left[\frac{1}{2}\left(-i\frac{d}{dx} - A(t)\right)^2 - \frac{1}{\sqrt{x^2+2}}\right]\Psi(x,t)$$

$$= \left[-\frac{1}{2}\frac{d^2}{dx^2} - i\frac{d}{dx}A(t) + \frac{A(t)^2}{2} - \frac{1}{\sqrt{x^2+2}}\right]\Psi(x,t)$$

$\frac{1}{\sqrt{x^2+2}}$ one-dimensional model potential,
in 1d, we must avoid the singularity at x=0
not so in 3d (no further explanations here)

$\Psi(x,t_0)$ Initial condition
Quantum language: initial “state” of the atom
at some time t_0 before the laser arrives

So, here is what we will do in C++:

- Describe wave function on the computer
- Get its time-evolution
- Display it appropriately

Lecture 1

A quick rundown of C++

Following https://www.w3schools.com/

Recommended references for C++ (exact definitions and properties)

https://cplusplus.com

https://en.cppreference.com

Connect to one of these sites if they appear for your google search

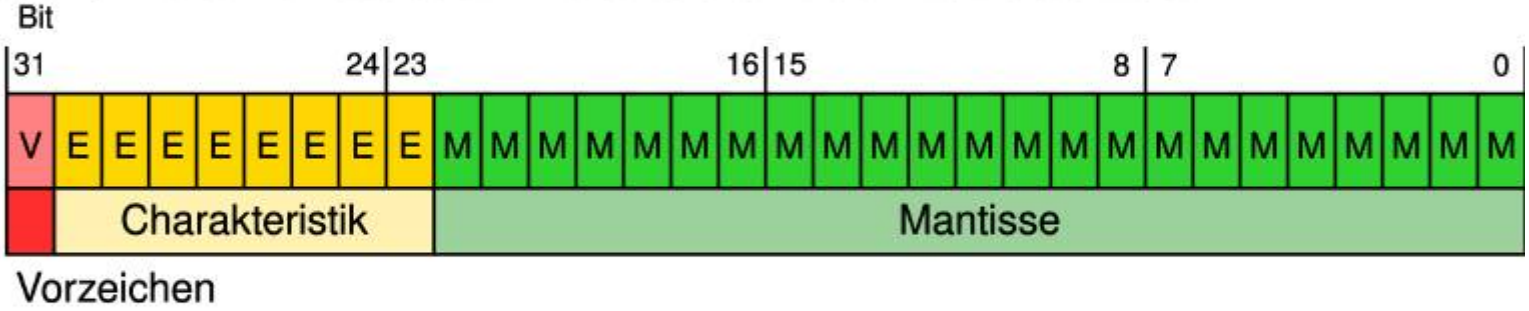
e.g. google search std::vector

Floating point numbers

Eine Dezimalzahl mit endlich vielen Stellen kann als Paar zweier ganzer Zahlen (Exponent,Mantisse) dargestellt werden:

$$12.345 = \overbrace{12345}^{\text{Mantisse}} \times \underbrace{10}_{\text{Basis}}^{\overbrace{-3}^{\text{Exponent}}} \tag{1}$$

Dafür existieren Industriestandards, z.B. IEE 754 wo von 32 Bit 8 für den Exponenten und 23 Bit für Mantisse und 1 Bit für das Vorzeichen verwendet werden:



Damit ist die Maximalzahl der Dezimalstellen auf ca. 7 beschränkt und dezimale Exponenten bis ± 38 können dargestellt werden. Die 38 ergibt sich, weil nicht der dezimale, sondern der binäre Exponent verwendet wird $2^{\pm 127} = 10^{\pm 127 \log 2} \approx 10^{38.2}$.

Der IEE 754 für 64 Bit floating point Zahlen nutzt 11 Bit für den Exponenten, womit sich ca. 15 bis 16 Dezimalstellen und exponenten bis ca. 308 darstellen lassen.

Dies sind auch die Grössenordnungen die sie für `float` (32 Bit) und `double` (64 Bit) auf üblicher Hardware erwarten können.

Reference und Pointer

Reference

Position einer Variablen im Memory – Adresse
Ab der Adresse gehören eine feste Anzahl Bytes der Variablen
Der Datentyp der Variablen bestimmt, wie die Bytes zu interpretieren sind

[Illustration]

Pointer

Enthält eine Adresse und (Compile-time) Info über den Datentyp der Adresse

Algebra mit Pointern: [Illustration]

In aktuellem C++ KEINE “raw” Pointer verwenden

Stattdessen:
std::shared_ptr<...>
std::unique_ptr<...>

Functions

Overloading

Beachten Sie:

Unterscheidung nur Anhand der Argumente

Nicht anhand des return-values

Scope

Variable die innerhalb der Funktion **deklariert** werden
haben nur dort definierten Sinn
Beim Verlassen der Funktion **wird das zugehörige Memory freigegeben**

Allgemeiner:
Variable die **innerhalb von { … }** deklariert werden haben nur dort Bedeutung

class – das Equivalent von (mathematischen) Begriffen

Vector aus einem Vektorraum über den reellen Zahlen

```
class Vector a,b,c;
double alfa;
```

folgende Operationen müssen definiert sein:

```
c = a + b;
c = alfa * a;
```

HilbertVec aus einem Hilbertraum

ist ein Vector wo ausserdem noch ein Skalarprodukt definiert ist:

```
class HilbertVec ha,hb;
double res;
res = ha.scalarProduct(hb)
```

Deklaration:

```
class HilbertVec: public Vector {
void scalarProduct(const HilbertVec& Rhs){
        … Berechnung des Skalarprodukts ...
};
};
```

public und private (und protected)

public

jeder, der das Objekt hat, kann auf die
“public members” (variable, funktionen) zugreifen

private

nur das Objekt selbst kann auf die
“private members” (variable, funktionen) zugreifen

Alles was nicht public sein muss, muss private (oder protected) sein

“private” definiert internen Zustand des Objekts

!!! Minimierung (oder genaue Kontrolle der verfügbaren Info !!!

Polymorphismus: virtual member function

```
class Rectangle {
protected:
      const double m_long, m_short;
public:
      Rectangle(double Long, double Short): m_long(Long), m_short(Short){};
      virtual double area() { return m_long*m_short;}
 }


class Rectangle: public Rectangle {
public:
      Rectangle(double Side): Rectangle(Side, Side) {};

      // silly way of implementing this...
      double area() { return std::pow(m_short,2);}

      // can access the protected variable members
      void print() const {std::cout<<m_long<<" "<<m_short;}
}
```

Polymorphismus: abstract base class

```
class Figure{
public:
      virtual double area() const=0;
}


class Circle: public Figure {
      double m_radius;
public:
      Circle(double Radius){m_radius=Radius;}
      double area() const { return std::pow(m_radius,2)*3.1415927;}
 }


class Rectangle: public Figure {
      const double m_long,m_short;
public:
      Rectangle(double Long, double Short):m_long(Long),m_short(Short){};
      double area() const { return m_long*m_short;}
 }
```

Verwendung von Polymorphismus

(siehe code/examples)

Programmierprinzip: Minimieren der Information

Nichts speichern was (leicht) aus vorhandener Info berechenbar ist

Nie Information (Variable) **duplizieren**

Nie Code **duplizieren** (kein cut & paste!)

Keine Information zugänglich machen, die “niemanden etwas angeht”
konsequente Nutzung von public / private / protected

Meine Lieblings-Container

std::vector< ... >

Complexe Werte von Koeffizienten

```
std::vector<std::complex<double>> coefficients;
```

std::set< ... >

Liste vorhandener Namen

```
std::set<std::string> names;
names.insert(“erwin”);
names.insert(“liese”);
names.insert(“erwin);
for(auto n: names)std::cout<<” “<<s; // gibt: erwin liese (oder liese erwin
```

std::map< ..., ... >

```
std::map<std::string, HilbertVector> all;
all[“ground state”]=Psi;
all[“excited state”]=OtherPsi;
all[“ground state”].scalarProduct(all[“excited state]);
```

Information zu std::vector auf cplusplus.com

<vector>

std::vector

```
template < class T, class Alloc = allocator<T> > class vector; // generic template
```

Vector

Vectors are sequence containers representing arrays that can change in size.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with ***amortized constant time*** complexity (see push_back).

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and forward_lists.

Analoge Info für

std::map

std::set

etc.

Meine Lieblings-Algorithmen aus <algorithms>

std::sort

suchen im Container

```
auto sorted(coeffiecients)
std::sort(sorted.begin(),sorted.end());
```

Info zur Skalierung
d.h.
Rechenzeit als Funktion der Anzahl der Elemente auf

cppreference.com
cplusplus.com

std::swap

```
std::vector<int> a={1,2,3}, b={5,6,7,8,9};
std::swap(a,b);
```

std::find

```
auto it=std::find(names.begin(),names.end(),”marie”);
if(it==names.end())std::cout<<”not found in names”;
```

… und Varianten …

Wo man Hilfe findet...

Welche Befehle und Datanstrukturen gibt es?

www.w3schools.com - C++ Reference: useful subset

Gibt es nicht … (ihr Wunsch)?

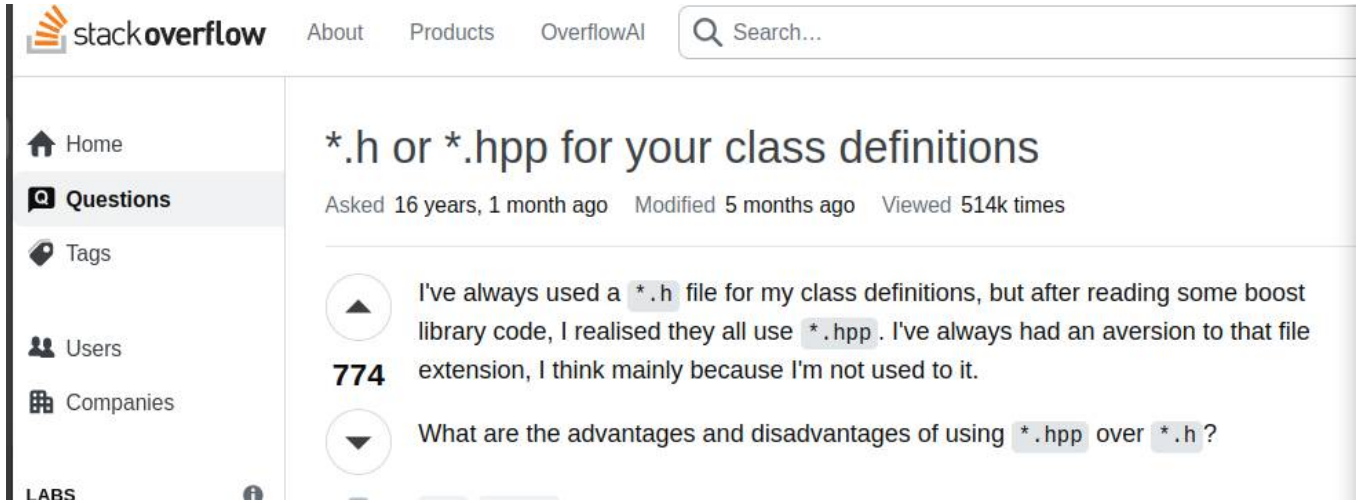
google….
gut nutzbar: “stackexchange”
manche Dinge gibt es, erstaunlicherweise, nicht in C++

Mysteriöse Probleme beim Compilieren

Compiler messages lesen lernen…
Klassiker: probleme mit “virtual table” – google
Text der Fehlermeldung in google
Text der Fehlermeldung in ChatGPT o.Ä.

Apropos...

“recently” on stackoverflow



one finds, among others:

2 @Christophe I wouldn't say the standard "unambiguously" recommends *.h. They leave it open to project convention: `SF.1: Use a .cpp suffix for code files and .h for interface files if your project doesn't already follow another convention` – PaulD Jul 29, 2022 at 16:15

Wenn’s wichtig ist, solche Antworten unbedingt nachprüfen!

Erwin

der Schrödinger Solver

```cpp
int main()
{
    std::cout<<"this is Erwin"<<std::endl;

    std::string inputFile("input");

    // ensure uniform input (and uniform input processing)
    // InputRead inp(inputFile);

    // ensure uniform output
    // OutputPrint out(inp);

    // create discrete representation of wave function
    // Discretization disc(inp);

    // set up the laser field
    // LaserField laser(inp);

    // set up the time-evolution
    // TimeEvolution evolution(inp);

    // initial state of the time-evolution
    // InitialState initialState(inp);
```

Liste der Aufgaben in Form eines main () {...}

```
int main(){

    // ensure uniform input (and uniform inpu
    // InputRead    np(inputFile);

    // ensure uniform output
    // OutputPrint  ut(inp);

    // create d    rete representation of wav
    // Discretization  isc(inp);

    // set up the laser field
    // LaserField  aser(inp);

    // set up the time-evolution
    // TimeEvolution  volution(inp)

    // initial state of the time-ev
    // InitialState  nitialState(inp);

    // out.info("input finished");

    // create the hamiltonian operator
    // OperatorHamiltonian  amiltonian(disc,laser);

    // out.print(hamiltonian.str());

    // WaveFunctionWithTime wf(initialState.compute(evolution,hamiltonian));

    // out.print("initial state\n"+wf.info());

    // evolution.propagate(wf,out);

    // out.info("done");

    return 0;
}
```

Aufgabengruppen in Form von class'es

Definiert input File, enthält allen aktuellen Input

Sorgt für einheitlichen, gut lesbaren Output

Wir beginnen hier

"Diskretisierung" – Darstellung der Wellenfunktion
zentrale Klasse!

Laserfeld mit Eigenschaften: waveLength(), intensity(), etc.

Zeitentwicklung: tBegin(), method(), propagate(), etc.

Anfangszustand: Grundzustand, Wellenpaket, etc.

Weitere zentrale Klasse: definiert das spezifische System

Diskretisierung - Mathematik

Approximation der komplexwertigen Wellenfunktion
als Linearkombination vorgegebener Funktionen

– “Basis” im Hilbertraum $h_{nk}(x)$ –

$$\Psi(x,t) \quad = \quad \sum_{n=0}^{N-1} \quad \sum_{k=0}^{K_n-1} \quad h_k^n(x) \quad c_k^n(t)$$

Approximation: suche Lösung nur auf Interval $x \in [x_0, x_N]$

Diskretisierung – die “Basis” $h^n_k(x)$

$$\Psi(x,t) \quad = \quad \sum_{n=0}^{N-1} \quad \sum_{k=0}^{K_n-1} \quad h_k^n(x) \quad c_k^n(t)$$

Approximation: suche Lösung nur auf Interval $x \in [x_0, x_N]$

Teilintervalle $[x_n, x_{n+1}]$

K_n Funktionen $h_k^n(x)$ auf Teilinterval $[x_n, x_{n+1}]$

Art der Funktionen: unterschiedlich…
wir wählen rellwertige Lagrangepolynome

Basis ist von uns gewählt, vorgegeben

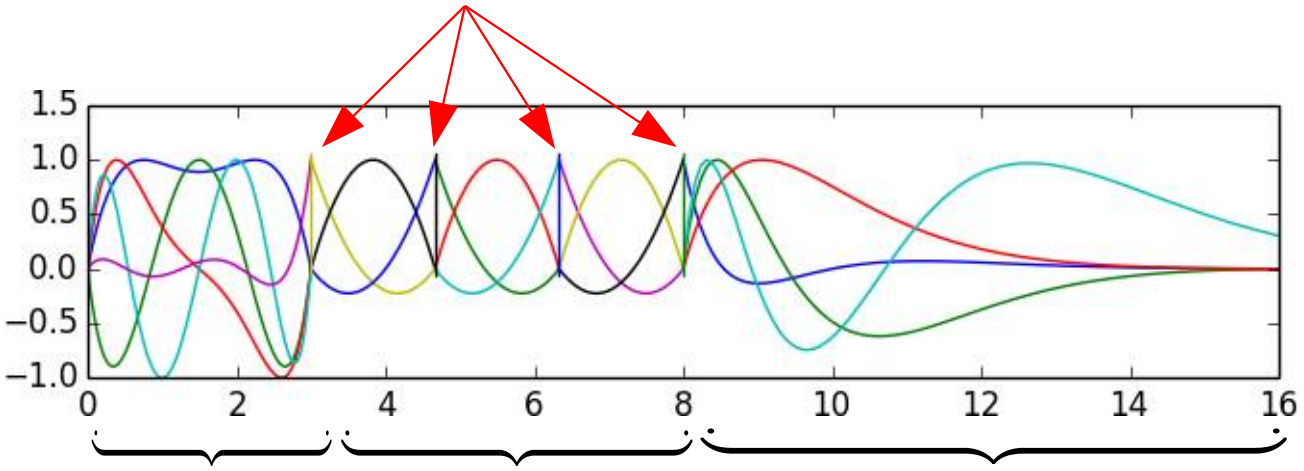
Die Basis $h_k^n(x)$ definiert den Raum, in dem wir nach Lösungen suchen

Approximation by piece-wise analytic functions (mostly polynomials)

Element boundaries

Functions continuous

Derivatives discontinuous



Polynomials Degree 5

Polynomials Degree 2

exp(-ar) x polynomials extends to r = ∞

Flexible, can be adjusted to local properties of the solution

Komplexwertige Koeffizienten $c^n_k(t)$

$$\Psi(x,t) \quad = \quad \sum_{n=0}^{N-1} \quad \sum_{k=0}^{K_n-1} \quad h^n_k(x) \quad c^n_k(t)$$

$c^n_k(t)$ tragen die konkrete Form und die Zeitabhängigkeit von $\Psi(x,t)$

$\Psi(x,t)$ komplexwertig, $h^n_k(x)$ reellwertig $\longrightarrow c^n_k(t)$ komplexwertig

Ordne alle $c^n_k(t)$ in (langen) komplexwertigen Vektor

$$\vec{c} = \begin{pmatrix} c^0_0 \\ \vdots \\ c^0_{K_0-1} \\ c^1_0 \\ \vdots \\ c^1_{K_1-1} \\ c^1_0 \\ \vdots \\ \text{usw.} \end{pmatrix}$$

Lineare Algebra

mit den Vektoren $\vec{c}$

(und mit der Hamiltonmatrix $\hat{H}$)

Zeitabhängige Schrödingergleichung

...in diskretisierter Form

$$i\hbar \frac{d}{dt}\vec{c}(t) = \hat{H}\vec{c}(t)$$

Gleichung für den diskreten Vektor $\vec{c}$ mit der Hamiltonmatrix $\hat{H}$

oder, allgemeiner:

$$i\hbar \frac{d}{dt}\vec{c}(t) = \hat{S}^{-1}\hat{H}\vec{c}(t)$$

$\hat{S}^{-1}$ ...Inverse einer “metrischen Matrix” $\hat{S}$

...Dinge werden später erklärt soweit für die Programmierung nötig...

Rekursive Struktur der $\vec{c}$

"Der Vektor $\vec{c}$ ist ein Vektor von Vektoren"

$$\vec{c} = \begin{pmatrix} c_0^0 \\ \vdots \\ c_{K_0-1}^0 \\ c_0^1 \\ \vdots \\ c_{K_1-1}^1 \\ c_0^1 \\ \vdots \\ \text{usw.} \end{pmatrix} = \left(\begin{array}{c} c_0^0 \\ \vdots \\ c_{K_0-1}^0 \\ \hline c_0^1 \\ \vdots \\ c_{K_1-1}^1 \\ \hline c_0^1 \\ \vdots \\ \text{usw.} \end{array}\right)$$

Definiere Teilvektor $\vec{c}^{\,n}$

$$\vec{c}^{\,n} := \begin{pmatrix} c_0^n \\ c_1^n \\ c_2^n \\ \vdots \\ c_{K_n-1}^n \end{pmatrix}$$

$$\vec{c} = \begin{pmatrix} \vec{c}^{\,0} \\ \vec{c}^{\,1} \\ \vdots \\ \vec{c}^{\,n-1} \end{pmatrix}$$

Rekursive Struktur in C++ (Standard Library)

std::vector<std::vector<std::complex<double>>> coefficients

oder auch: class Coefficients mit rekursiven Eigenschaften...

class Discretization

Rekursive Definition

"Diskretisierung" der rellen Achse
durch N Teilintervalle

Diskretisierung jedes Teilintervalls n
durch Funktionen $h^n_k(x)$, $k=0,\ldots,K_n-1$

Beides wird durch die gleiche class Discretization dargestellt

Screenshot...

```
class Discretization
{
    // a Discretization has a vector of sub-discretization
    std::vector<std::shared_ptr<Discretization>> m_child;

    // the "Basis" will be:
    //      on the level of the real axis:
    //          class BasisFE - numbering the intervals [ x[n], x[n+1] ]
    //      on each interval [ x[n], x[n+1] ]:
    //          class BasisLagrange implementing Lagrange polynomials h[n,k]
    std::shared_ptr<Basis> m_basis;

    // constructor for recursive construction
    Discretization(InputRead & Inp, Discretization & Parent);

public:
    // Inp must provide parameters for
    // - construction of m_basis
    // - construction of sub-discretizations m_discr
    Discretization(InputRead & Inp);
};
```

Beachte: nur 4 Zeilen wirklicher Code...

Programmierprinzipien: be simple!

Programmieren Sie nicht für alle Eventualitäten,

programmieren Sie für das konkrete Problem!

Der skizzierte Code verstösst gegen dieses Prinzip:

good: eignet sich hervorragend, um in 3 und mehr Dimensionen zu gehen
bad: ist unnötig abstrakt für den harmonischen Oszillator in 1d

Empfehlung: Herangehen an ein Problem

Zuerst allgemeine Struktur definieren

- class'es, dh. Begriffe wie "Vector", "WaveFunction" etc.
- ihre "inneren Eigenschaften": welche Werte braucht es zur Definition
- ihre Funktionen, z.B. size(), value(), scalarProduct(), propagate()

Danach erst konkrete Algorithmen

- berechne das a.scalarProduct(b) von Vector a,b;
- propagate(psi,t0,t1) die Wellenfunktion psi von t_0 bis t_1

class DiscretizationSimple and more...

$$\Psi(x,t) \quad = \quad \sum_{n=0}^{N-1} \quad \sum_{k=0}^{K_n-1} \quad h_k^n(x) \quad c_k^n(t)$$

List of the basis sets of abstract class Basis

class BasisLagrange – implements Basis

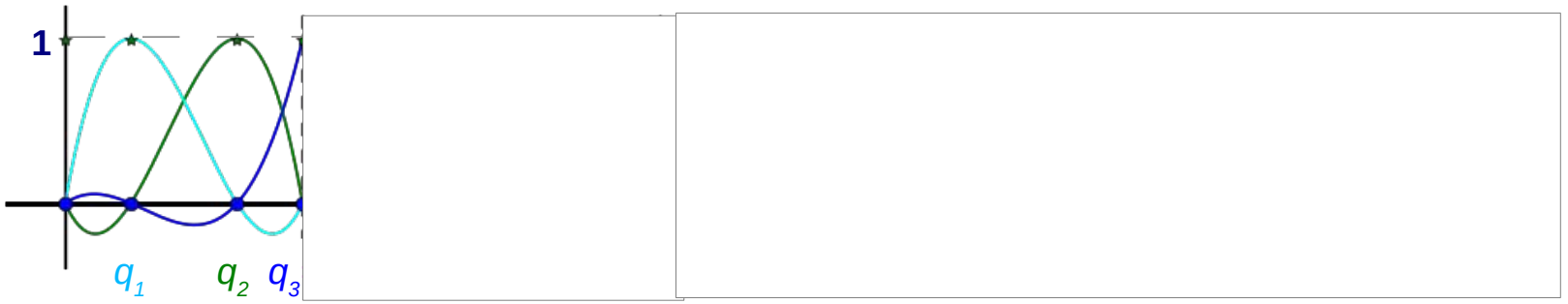
class WaveFunctionSimple $\Psi(x,t)$

- is a **VectorHilbert**: Vector + scalar product
- in addition it has a time()
- it is constructed by
  creating the a vector according to DiscretizationSimple
  setting its Time

DiscretizationSimple(std::shared_ptr<DiscretizationSimple> Disc, double Time=0)

Lagrange-Polynome $h_k^n(x)$

Gegeben Punkte q_0, q_1, q_2, q_3
4 Polynome, je auf einem Punkt = 1, auf den anderen =0



analog auf den intervallen $[q_4=q_3,q_6]$ und $[q_7=q_6,...$

Punkte im Prinzip frei wählbar aber...

Spezielle Wahl der Punkte q_i für effiziente numerische Integration

Gauss-Lobatto Quadraturpunkte

DVR – "discrete variable representation"
d.h. Lagrangepolynome mit diesen speziellen q_i

Guide to topics

Programming Style

Eigen Library

Constructors

Overload of +,=, () etc.

return *this

Static members

std::shared_ptr<...>

std::function

lambda-expressions

Programming style

Name conventions (for this course)

- camelCase (not snake_case)
- m_memberData
- localVariable
- ParameterVariable

Constructor

constructor overload

recursive use of constructor

vectorHilbert.h: std::initializer_list for intializing with {val0,val1,val2,...}

Overload +,-,+=, () etc.

vectorHilber.h: VectorHilbert& operator+=(const VectorHilbert & Y);

Return a reference to the object

vectorHilbert.cpp:

```
VectorHilbert & VectorHilbert::operator+=(const VectorHilbert & Rhs){
    .....
    ....
    return *this;
}
```

Static members

Functions

vectorHilbert.h: static void test();

discretizationSimple.h: static void test(InputRead &Inp);

Variables

basisDVR.h: static std::map<int,QuadratureRule> m_lobatto;

initialized in

basisDVR.cpp: std::map<int,BasisDVR::QuadratureRule> BasisDVR::m_lobatto=

Using std::shared_ptr<...>

discretizationSimple.cpp

```
m_basis.push_back(std::make_shared<BasisDVR>(bound[n-1],bound[n],order));
```

std::function – Functions as variables

discretizationSimple.h:

```
VectorHilbert vector(std::function<std::complex<double>(double)> Func) const;
```

discretizationSimple.cpp:

```
VectorHilbert DiscretizationSimple::vector(std::function<std::complex<double>(double)> Func) const{
   ...
        res.push_back(Func(bDVR->quadPts()(k)));
   ...
```

lambda-expressions

Eigen – a linear algebra template library

VectorXcd … complex<double> vector
VectorXd … double vector

Map<VectorXd>(pointer, size) … map array, vector without any speed loss
done at compile time!